# Python Notes

Join Telugu Data Science Community:
https://nerchukoacademy.graphy.com/courses/Data-Science-Community-674175349d31e57fc0e5a87f-674175349d31e57fc0e5a87f

> https://youtu.be/wC7pbJZB1MU?si=ohfnNtltiN6T8olw

## Introduction to Python and Google Colab

1. **Google Colab Overview**:
   - Google Colab is a cloud-based platform that functions like a Jupyter Notebook.
   - No installation is required, and it allows you to write and run Python code directly in your browser.
   - Files can be uploaded and accessed locally for use in your notebooks.
2. **Arithmetic Operations in Python**:
   - **Basic Operations**:
     - `+`: Addition (e.g., `2 + 3 = 5`)
     - `-`: Subtraction (e.g., `2 - 1 = 1`)
     - `*`: Multiplication (e.g., `2 * 1 = 2`)
     - `/`: Division (returns a float, e.g., `3 / 2 = 1.5`)
   - **Advanced Operators**:
     - `//`: Floor Division (e.g., `3 // 2 = 1`)
     - `%`: Modulo (remainder of division, e.g., `5 % 3 = 2`)

    - `*`: Exponentiation (e.g., `2 ** 3 = 8`, which is $2^3$)
3. **Order of Operations in Python**:
    - Follows **BODMAS** (Brackets, Orders, Division/Multiplication, Addition/Subtraction).
    - Example: `2 + 10 * 1` results in `12` because multiplication is done before addition.
4. **Floating Point Numbers and Roots**:
    - Division often produces floating-point results (e.g., `3 / 2 = 1.5`).
    - Square root can be calculated using exponentiation (e.g., `2 ** 0.5 = 1.414`).
5. **Using Comments in Python**:
    - Use `#` for single-line comments to make code more understandable.

## Variables in Python

1. **Naming Variables**:
    - Variable names **cannot** start with a number (e.g., `1variable` is invalid).
    - Variable names **cannot** contain spaces. Use an underscore (`_`) to separate words (e.g., `my_variable`).
    - Avoid using **Python reserved keywords** like `list` or `string` for variable names.
2. **Declaring Variables**:
    - Variables are declared by simply assigning a value (e.g., `a = 5`).
    - Python supports dynamic typing, so you can assign different data types to a variable over time.
    - **Boolean Values**: You can assign a boolean value `True` or `False` (e.g., `is_active = True`).
3. **Syntax Error Example**:
    - Invalid variable names (e.g., `number 1 = 5`) will raise a **syntax error**.

- **Correct syntax**: Use underscores (e.g., `number_1 = 5`).

---

## Boolean Data Type

1. **Boolean Values**:
   - `True` and `False` are the two possible boolean values.
   - **Common Issue**: `True` is a built-in keyword in Python, so using `True` as a variable name will result in a **NameError**. Always define it as `is_active = True` instead of reusing the keyword.

---

## Simple Interest Calculation Program

1. **Formula**:

   Where:

   SI=P×T×R/100

   - `P` is the principal amount.
   - `T` is the time in years.
   - `R` is the rate of interest.

## Python Code:

```
# Function to calculate simple interest
def calculate_simple_interest(principal, time, rate):
    # Formula to calculate Simple Interest
    simple_interest = (principal * time * rate) / 100
    return simple_interest

# Example data
P = 10000  # Principal amount in INR
T = 5      # Time period in years
R = 10     # Rate of interest in percentage

# Calculate Simple Interest
```

```
SI = calculate_simple_interest(P, T, R)

# Output the result
print(f"Principal: {P} INR")
print(f"Time: {T} years")
print(f"Rate of Interest: {R}%")
print(f"Simple Interest: {SI} INR")
```

## Example Calculation:

For the following values:

- **Principal (P)** = 10,000 INR
- **Time (T)** = 5 years
- **Rate of Interest (R)** = 10%

Using the formula:

$SI = 10,000 \times 5 \times 10 / 100 = 500$

So, the Simple Interest (SI) will be **500 INR**.

## Program Output:

```
Principal: 10000 INR
Time: 5 years
Rate of Interest: 10%
Simple Interest: 500 INR
```

This program defines a function `calculate_simple_interest()` which takes the principal amount, time, and rate of interest as inputs and calculates the simple interest using the formula.

---

## Dynamic Typing in Python

1. **What is Dynamic Typing?**:
    - Python allows reassigning variables to different data types. For example, a variable initially assigned as an integer can later hold a string or a list.

2. **Example of Dynamic Typing**:
    - `var = 5` (integer)
    - `var = "hello"` (string)
    - Python adjusts the type automatically without needing explicit type declarations.
3. **Checking Variable Type**:
    - Use the `type()` function to check the type of a variable (e.g., `type(var)`).
4. **Data Types**:
    - Common data types include:
        - **int**: Integer numbers (e.g., `5`)
        - **float**: Decimal numbers (e.g., `5.5`)
        - **str**: String (e.g., `"hello"`)
        - **list**: List (e.g., `[1, 2, 3]`)
        - **dict**: Dictionary (e.g., `{"key": "value"}`)
        - **set**: Set (e.g., `{1, 2, 3}`)
        - **bool**: Boolean (e.g., `True` or `False`)

## Working with Data Types in Python

1. **Boolean Data Type (`bool`)**:
    - Boolean values can be either `True` or `False`.
    - Python allows reassigning variables to different data types, making it flexible in handling various types like integers, floats, and strings.
2. **Dynamic Typing**:
    - Python supports **dynamic typing**, which means a variable's data type can be changed after it has been defined. For example:

```
my_variable = 5  # integer
my_variable = "Hello"  # string
```

3. **Checking Data Types**:
   - The `type()` function can be used to check the data type of a variable:

```
print(type(my_variable))  # prints <class 'str'> if my_
variable is a string
```

---

## Strings in Python

1. **Creating Strings**:
   - Strings in Python are sequences of characters and can be created using either **single quotes** or **double quotes**:

```
string1 = 'Hello'  # using single quotes
string2 = "Hello"  # using double quotes
```

2. **Advantages and Disadvantages of Quotes**:
   - **Single quotes** are great for creating simple strings, but they have a limitation when you need to include an apostrophe (`'`) inside the string. To solve this, you can either use **double quotes** for the string or **escape the apostrophe** with a backslash (`\`):

```
string_with_apostrophe = "It's a beautiful day"  # doub
le quotes used
string_with_apostrophe = 'It\'s a beautiful day'  # esc
ape character used
```

3. **Backslash (`\`) as an Escape Character**:
   - When you need to include special characters in a string, like quotes or newlines, you use the backslash (`\`):

```
string_with_quotes = "He said, \"Hello!\""  # using bac
kslash for double quotes inside a string
```

4. **Printing Strings**:
   - You can print a string using the `print()` function:

```
print("Welcome to my channel!")  # prints the string to
the console
```

## Working with Strings and Indexing in Python

1. **Printing with New Line Characters (`\n`)**:
   - In Python, you can use the **new line character (`\n`)** to print text on a new line. It is often used to format output:

```
print("Welcome to my channel.\nLike this video.")  # Th
e text after '\n' will be printed on a new line.
```

2. **Checking String Length (`len()` Function)**:
   - To find the length of a string (i.e., the number of characters including spaces), you can use the built-in `len()` function:

```
my_string = "Welcome to my channel"
print(len(my_string))  # Outputs the length of the stri
ng, which is 22
```

3. **String Indexing**:
   - Python uses **zero-based indexing** for strings, meaning that the first character of the string has an index of 0, the second has an index of 1, and so on.

- You can access individual characters of a string using **square brackets (`[]`)** with the index value:

```
my_string = "Hello"
print(my_string[0])  # Prints 'H' (the first character)
print(my_string[1])  # Prints 'e' (the second characte
r)
print(my_string[2])  # Prints 'l' (the third character)
```

4. **Negative Indexing**:
   - Python also supports **negative indexing**, which allows you to access the characters from the end of the string. For example, the index `-1` corresponds to the last character, `-2` to the second-to-last character, and so on:

```
print(my_string[-1])  # Prints 'o' (the last character)
print(my_string[-2])  # Prints 'l' (the second-to-last
character)
```

5. **String Slicing**:
   - You can extract a substring from a string using **slicing**. Slicing involves specifying a **start index**, an **end index**, and optionally a **step** (with a colon):

```
my_string = "Hello"
print(my_string[0:2])  # Prints 'He' (from index 0 to i
ndex 1)
print(my_string[1:4])  # Prints 'ell' (from index 1 to
index 3)
print(my_string[::2])  # Prints 'Hlo' (every second cha
racter)
```

6. **String Indexing Example**:

- You can also use **indexing** to access specific characters and modify how you print strings:

```
my_string = "Hello"
print(my_string[0])  # Prints 'H' (first character)
print(my_string[1])  # Prints 'e' (second character)
print(my_string[2])  # Prints 'l' (third character)
print(my_string[3])  # Prints 'l' (fourth character)
```

## String Slicing:

In Python, **slicing** allows you to extract a specific part of a string. The general syntax for slicing is:

```
string[start_index:end_index]
```

- **start_index**: This is the index from where you want to start the slice.
- **end_index**: This is the index where the slice ends (but the character at `end_index` is not included in the slice).

## Example of slicing:

```
s = "Hello"
print(s[0:3])  # Output: "Hel"
```

- Here, the slice starts from index 0 and ends at index 3 (but does not include index 3).
- The result is "Hel" (characters at indexes 0, 1, and 2).

If you mention `0:4`, it will return characters at indexes 0, 1, 2, and 3:

```
print(s[0:4])  # Output: "Hell"
```

## Grabbing All Values (Full String):

If you omit both `start_index` and `end_index`, the entire string will be returned:

```
print(s[:])  # Output: "Hello"
```

This grabs all values in the string, printing the full string.

## Negative Indexing:

Python also supports **negative indexing**, where `-1` represents the last character of the string, `-2` represents the second-last character, and so on.

## Example of negative indexing:

```
s = "Hello"
print(s[-1])  # Output: "o" (last character)
print(s[-2])  # Output: "l" (second last character)
```

## String Methods (Uppercase and Lowercase):

You can convert a string to uppercase or lowercase using string methods like `.upper()` and `.lower()`.

## Example of converting to lowercase:

```
s = "Hello World"
print(s.lower())  # Output: "hello world"
```

## Example of converting to uppercase:

```
s = "Hello World"
print(s.upper())  # Output: "HELLO WORLD"
```



## String Methods:

1. `lower()` and `upper()`:

- The `.lower()` method converts a string to **lowercase**, and `.upper()` converts it to **uppercase**.
- For example:

```
s = "Hello World"
print(s.lower())  # Output: "hello world"
print(s.upper())  # Output: "HELLO WORLD"
```

2. `split()`:
   - The `.split()` method splits a string into a list of substrings based on a delimiter.
   - By default, it splits at spaces.

```
s = "Hello World"
print(s.split())  # Output: ['Hello', 'World']
```

   - You can specify a custom delimiter inside the `.split()` method:

```
s = "Hello World"
print(s.split('o'))  # Output: ['Hell', ' W', 'rld']
```

## String Formatting:

In Python, there are several ways to format strings, especially when combining non-string values like numbers or lists with strings. There are **three main methods** for string formatting:

1. **Modulo (`%`) Operator**:
   - This method is older but still used in some cases. You use placeholders like `%s` for strings, `%d` for integers, `%f` for floating-point numbers, etc.
   - Example:

```
name = "Nandi"
age = 25
```

```
print("My name is %s and I am %d years old." % (name, a
ge))
# Output: My name is Nandi and I am 25 years old.
```

2. `format()` **Method**:
   - Introduced in Python 2.7 and Python 3.0, the `format()` method provides a more flexible way to format strings.
   - Example:

```
name = "Nandi"
age = 25
print("My name is {} and I am {} years old.".format(nam
e, age))
# Output: My name is Nandi and I am 25 years old.
```

3. **f-Strings (formatted string literals)**:
   - Available in Python 3.6 and later, f-strings provide the most readable and concise way to format strings by embedding expressions directly inside curly braces `{}`.
   - Example:

```
name = "Nandi"
age = 25
print(f"My name is {name} and I am {age} years old.")
# Output: My name is Nandi and I am 25 years old.
```

## Example with Input and Formatting:

You also mentioned using **user input** in string formatting, so here's an example:

```
name = input("Enter your name: ")  # User types 'Nandi'
age = input("Enter your age: ")  # User types '25'
```

```
# Using string formatting
print(f"Hello {name}, you are {age} years old.")
```

This will display the formatted string with the user-provided inputs.

## Summary:

- `.lower()` converts a string to lowercase, and `.upper()` converts it to uppercase.
- `.split()` splits the string into a list based on a delimiter.
- String formatting in Python can be done using `%`, `.format()`, or f-strings.

## Lists in Python

1. **Creating Lists**:
   - Lists are created using square brackets `[]`:

```
my_list = [1, 2, 3, 4, 5]
```

2. **Accessing List Elements**:
   - Elements can be accessed using indexing (starting from 0):

```
print(my_list[0])  # prints 1
print(my_list[-1]) # prints 5 (last element)
```

3. **Slicing Lists**:
   - You can slice lists to retrieve sublists. The general syntax is:

```
my_list[start:end]  # Retrieves elements from index 'st
art' to 'end-1'
```

   - Examples:

```
print(my_list[1:4])  # prints [2, 3, 4]
print(my_list[:3])   # prints [1, 2, 3]
```

```
print(my_list[2:])   # prints [3, 4, 5]
```

4. **Using `len()` with Lists**:
   - You can get the length of a list using `len()`:

```
print(len(my_list))  # prints 5
```

5. **Empty Lists**:
   - You can create an empty list with `[]`:

```
empty_list = []
```

## Miscellaneous

- **Indexing in Python**: You explained that Python lists are indexed starting from 0, and negative indices are used to access elements from the end of the list.
- **Boolean and Mixed Data Types in Lists**: Python lists can contain elements of different data types, including integers, strings, booleans, and even other lists.

## List Operations:

1. **Indexing**:
   - Lists can be indexed using square brackets `[]`. The starting index is 0, and you can also use negative indexing to start from the end (`1` for the last element).
   - Example: `my_list[1]` gives the second element.

```
my_list = [10, 20, 30, 40, 50]
# Accessing the second element (index 1)
print(my_list[1])  # Output: 20
```

```
# Negative indexing to access the last element
print(my_list[-1])  # Output: 50
```

2. **List Operations (Addition, Multiplication)**:
   - You can add lists using the `+` operator: `my_list + another_list`.
   - Multiplying a list by an integer will repeat its elements: `my_list * 2`.

```
my_list = [1, 2, 3]
another_list = [4, 5, 6]

# List addition (concatenation)
combined_list = my_list + another_list
print(combined_list)  # Output: [1, 2, 3, 4, 5, 6]

# List multiplication (repeating elements)
multiplied_list = my_list * 2
print(multiplied_list)  # Output: [1, 2, 3, 1, 2, 3]
```

3. **Modifying Lists**:
   - You can modify list elements by using their index, for example: `my_list[2] = 10` will change the third element to `10`.

```
my_list = [10, 20, 30, 40]

# Modifying the third element (index 2)
my_list[2] = 100
print(my_list)  # Output: [10, 20, 100, 40]
```

4. **List Methods**:
   - **append()**: Adds an element to the end of the list.
   - **insert(index, value)**: Inserts an element at the specified index.
   - **remove(value)**: Removes the first occurrence of a specified value.

- **pop()**: Removes and returns the last element (or an element at a specific index if provided).
- **reverse()**: Reverses the list in place.

```
my_list = [10, 20, 30, 40]

# append() - Adds an element at the end of the list
my_list.append(50)
print(my_list)  # Output: [10, 20, 30, 40, 50]

# insert() - Inserts an element at a specified index
my_list.insert(2, 25)  # Insert 25 at index 2
print(my_list)  # Output: [10, 20, 25, 30, 40, 50]

# remove() - Removes the first occurrence of the specified va
my_list.remove(30)
print(my_list)  # Output: [10, 20, 25, 40, 50]

# pop() - Removes and returns the last element
last_element = my_list.pop()
print(last_element)  # Output: 50
print(my_list)  # Output: [10, 20, 25, 40]

# reverse() - Reverses the list in place
my_list.reverse()
print(my_list)  # Output: [40, 25, 20, 10]
```

5. **Additional Functions**:
   - **len()**: Returns the number of elements in the list.
   - **max()**: Returns the maximum value in the list.
   - **min()**: Returns the minimum value in the list.

```
my_list = [10, 20, 30, 40, 50]
```

```
# len() - Returns the number of elements in the list
length = len(my_list)
print(length)  # Output: 5

# max() - Returns the maximum value in the list
maximum = max(my_list)
print(maximum)  # Output: 50

# min() - Returns the minimum value in the list
minimum = min(my_list)
print(minimum)  # Output: 10
```

## Dictionary Basics:

In Python, a **dictionary** is a collection of **key-value pairs**, where each key is unique. The values associated with keys can be of any data type, and dictionaries are unordered collections, meaning that the items are not stored in a particular order.

- **Indexing and Keys**: You can access the value of a dictionary using the key.

## Examples:

## 1. Indexing and Keys

```
# Creating a dictionary
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Accessing values by key
print(my_dict["name"])  # Output: Alice
print(my_dict["age"])   # Output: 25
print(my_dict["city"])  # Output: New York
```

## 2. Using `in` and `not in`

- `in`: Checks if the key exists in the dictionary.
- `not in`: Checks if the key does not exist in the dictionary.

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Check if a key exists
if "age" in my_dict:
    print("True")  # Output: True
else:
    print("False")

# Check if a key does not exist
if "address" not in my_dict:
    print("Key not found")  # Output: Key not found
```

## 3. `get()` Method

The `get()` method is used to retrieve the value associated with a key. If the key does not exist, it returns `None` by default. You can also specify a default value to return if the key is not found.

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Using get() to fetch values
print(my_dict.get("name"))      # Output: Alice
print(my_dict.get("address"))   # Output: None

# Using get() with a default value if key is not found
print(my_dict.get("address", "Not Available"))  # Output: Not
Available
```

## Additional Dictionary Operations:

- **Adding/Updating Elements**: You can add new key-value pairs or update existing ones by assigning a value to a key.

```
# Adding a new key-value pair
my_dict["email"] = "alice@example.com"
print(my_dict)

# Updating an existing key-value pair
my_dict["age"] = 26
print(my_dict)  # Output: {'name': 'Alice', 'age': 26, 'cit
y': 'New York', 'email': 'alice@example.com'}
```

- **Removing Elements**: You can remove a key-value pair using the `del` statement or the `pop()` method.

```
# Using del to remove an element
del my_dict["city"]
print(my_dict)  # Output: {'name': 'Alice', 'age': 26, 'emai
l': 'alice@example.com'}

# Using pop() to remove and return the value of a key
email = my_dict.pop("email")
print(email)  # Output: alice@example.com
print(my_dict)  # Output: {'name': 'Alice', 'age': 26}
```

- **Dictionary Length**: Use the `len()` function to get the number of key-value pairs in a dictionary.

```
print(len(my_dict))  # Output: 2
```



## Tuples:

- **Immutability**: Tuples are similar to lists but cannot be modified once created.
- **Creating Tuples**: You create a tuple using parentheses `()`, unlike lists which use square brackets `[]`.

```
t = (1, 2, 3)  # Tuple creation
```

- **Indexing**: You can access elements of a tuple using indices, similar to lists. Tuples support negative indexing as well.
- **Operations**: While you cannot change the elements of a tuple, you can perform operations like repetition:

```
t = (1, 2, 3)
t = t * 2  # Results in (1, 2, 3, 1, 2, 3)
```

## Sets:

- **Unordered Collection of Unique Elements**: Sets do not allow duplicate elements. They can be created using curly braces `{}` or the `set()` function.

```
s = {1, 2, 3}  # Set creation
```

- **Adding Elements**: You can add elements to a set using the `.add()` method, but if you try to add a duplicate, it won't change the set.

```
s.add(4)
s.add(2)  # Won't change the set because 2 is already pres
ent
```

:

## Comparison Operators:

- `==`: Checks if two values are equal.
- `!=`: Checks if two values are not equal.
- `>`: Checks if the left value is greater than the right.
- `<`: Checks if the left value is less than the right.
- `>=`: Checks if the left value is greater than or equal to the right.

- `<=`: Checks if the left value is less than or equal to the right.

### Example:

```
x = 3
y = 2

print(x > y)  # True, because 3 is greater than 2
print(x < y)  # False, because 3 is not less than 2
print(x == y) # False, because 3 is not equal to 2
print(x != y) # True, because 3 is not equal to 2
```

### If-Else Statements:

In Python, `if` statements execute a block of code if a condition is `True`. If the condition is `False`, the code in the `else` block will be executed.

```
# Example 1: Simple if statement
number = 2
if number == 2:
    print("It was true")

# Example 2: If-Else Statement
name = "Nandi"
if name == "Nandi":
    print("Hi, Nandi!")
else:
    print("Who are you?")
```

### If-Elif-Else Statements:

You can use `elif` to check multiple conditions. If the `if` condition is `False`, the program will check the `elif` condition and so on. If all conditions fail, the code in the `else` block is executed.

```
# Example 3: If-Elif-Else Statement
age = 20
if age < 18:
    print("You're a minor.")
elif age >= 18 and age < 60:
    print("You're an adult.")
else:
    print("You're a senior.")
```

## Loops in Python

In Python, loops are used to iterate over a sequence (like a list, tuple, string, or range) or execute a block of code multiple times. Python supports two types of loops:

1. **For Loop**: Used to iterate over a sequence (like a list, tuple, or string).
2. **While Loop**: Used to repeat a block of code while a condition is `True`.

Let's explore both types of loops with detailed explanations and examples.

---

## 1. For Loop

The `for` loop is used to iterate over a sequence of elements (like a list or string). The loop runs once for each item in the sequence.

## Basic Syntax:

```
for variable in sequence:
    # Code to execute
```

Where:

- `variable`: A variable that takes the value of each item in the sequence.
- `sequence`: A collection (like a list or string) or a range.

## Examples:

- **For loop with a list**:

```
nums = [1, 2, 3, 4, 5]
for num in nums:
    print(num)
```

**Output:**

```
1
2
3
4
5
```

- **For loop with range()**:

The `range()` function is commonly used to generate a sequence of numbers, which can then be iterated over by a `for` loop.

```
for i in range(1, 6):  # From 1 to 5
    print(i)
```

**Output:**

```
1
2
3
4
5
```

- **For loop to print even numbers**:

```
for num in range(0, 11):  # Range from 0 to 10
    if num % 2 == 0:
```

```
        print(num)
```

**Output:**

```
0
2
4
6
8
10
```

## 2. Sum of N Natural Numbers Using For Loop

You can use a `for` loop to calculate the sum of the first `N` natural numbers.

```
n = int(input("Enter a positive number: "))  # User input
sum = 0
for i in range(1, n + 1):  # Loop from 1 to n
    sum += i  # Add i to sum
print(f"The sum of first {n} natural numbers is: {sum}")
```

**Example Input:**

```
Enter a positive number: 4
```

**Output:**

```
The sum of first 4 natural numbers is: 10
```

## 3. Break and Continue in For Loop

The `break` and `continue` statements allow you to control the flow of the loop.

- `break`: Exits the loop when a certain condition is met.
- `continue`: Skips the current iteration and moves to the next iteration.

### Example using `break` and `continue`:

```
for i in range(10):  # Loop from 0 to 9
    if i == 2:
        print("Skipping 2")
        continue  # Skip the rest of the loop for i == 2
    if i == 5:
        print("Breaking at 5")
        break  # Exit the loop when i == 5
    print(i)
```

**Output:**

```
0
1
Skipping 2
3
4
Breaking at 5
```

---

## 4. While Loop

The `while` loop repeats a block of code as long as the specified condition is `True`.

## Basic Syntax:

```
while condition:
    # Code to execute
```

Where:

- `condition`: A Boolean expression that is evaluated before each iteration. If it is `True`, the loop runs. If it is `False`, the loop stops.

## Example of a while loop:

```
x = 0
while x < 5:
    print(f"X is currently {x}")
    print("X is still less than 10, adding 1 to X")
    x += 1  # Increment x
```

**Output:**

```
X is currently 0
X is still less than 10, adding 1 to X
X is currently 1
X is still less than 10, adding 1 to X
X is currently 2
X is still less than 10, adding 1 to X
X is currently 3
X is still less than 10, adding 1 to X
X is currently 4
X is still less than 10, adding 1 to X
```

---

## 5. While Loop with Else

A `while` loop can have an `else` block, which is executed when the condition becomes `False` (i.e., the loop terminates).

```
number = 0
while number < 5:
    print(f"Current number is {number}")
    number += 1
else:
    print("The while loop has finished executing.")
```

**Output:**

```
Current number is 0
Current number is 1
```

```
Current number is 2
Current number is 3
Current number is 4
The while loop has finished executing.
```

## 6. Break Example in a While Loop

You can use the `break` statement to exit a `while` loop prematurely.

```
number = 0
while True:  # Infinite loop
    if number == 5:
        print("Breaking at 5")
        break  # Exit the loop when number is 5
    print(f"Current number is {number}")
    number += 1
```

**Output:**

```
Current number is 0
Current number is 1
Current number is 2
Current number is 3
Current number is 4
Breaking at 5
```



## Summary of Loop Control Statements:

- `for` **loop**: Iterates over a sequence (like a list, string, or range).
- `while` **loop**: Executes as long as the condition is `True`.
- `break`: Exits the loop immediately.
- `continue`: Skips the current iteration and continues with the next one.

- `else` **with loops**: Executes after the loop completes, unless interrupted by a `break` statement.

## Functions in Python

A **function** is a block of code that only runs when it is called. Functions allow you to group code into reusable blocks, which makes your code cleaner, more organized, and easier to maintain.

Python has both **user-defined functions** (defined by the programmer) and **built-in functions** (provided by Python). Let's explore both types of functions in detail.

---

## 1. User-Defined Functions

A **user-defined function** is a function that you define using the `def` keyword. It can take inputs (parameters), perform tasks, and return an output.

## Syntax of a User-Defined Function:

```
def function_name(parameters):
    # Code block
    # You can use parameters here
    return result  # optional
```

Where:

- `function_name`: The name of the function.
- `parameters`: The inputs to the function (optional).
- `return`: The value that the function outputs (optional). If no `return` is used, the function returns `None` by default.

## Examples:

1. **Function without parameters and return value**:

```
def greet():
    print("Hello, welcome to Python!")

greet()
```

**Output:**

```
Hello, welcome to Python!
```

1. **Function with parameters and return value**:

```
def add(a, b):
    return a + b

result = add(3, 5)
print(result)
```

**Output:**

```
8
```

1. **Function with default parameters**:

```
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet("Alice")  # Using the provided argument
greet()         # Using the default argument
```

**Output:**

```
Hello, Alice!
Hello, Guest!
```

1. **Function with multiple return values** (returning a tuple):

```
def calculate(a, b):
    sum_val = a + b
    diff_val = a - b
    return sum_val, diff_val  # Returns a tuple

result = calculate(10, 5)
print(result)  # (15, 5)
```

---

## 2. Built-in Functions

Python comes with a large set of built-in functions that are always available for use. These functions cover a wide range of tasks, such as mathematical operations, string manipulation, and more.

## Some Common Built-in Functions:

1. `print()`: Prints the specified message to the console.

```
print("Hello, world!")
```

**Output:**

```
Hello, world!
```

1. `len()`: Returns the length (number of elements) of an object like a string, list, or tuple.

```
text = "Hello"
print(len(text))  # 5
```

**Output:**

```
5
```

1. `range()`: Generates a sequence of numbers, often used with loops.

```
for i in range(1, 6):
    print(i)
```

**Output:**

```
1
2
3
4
5
```

1. `type()`: Returns the type of an object.

```
x = 5
print(type(x))  # <class 'int'>
```

**Output:**

```
<class 'int'>
```

1. `sum()`: Returns the sum of all elements in an iterable (like a list or tuple).

```
numbers = [1, 2, 3, 4]
print(sum(numbers))  # 10
```

**Output:**

```
10
```

1. `max()` **and** `min()`: Returns the maximum and minimum values from an iterable.

```
numbers = [1, 2, 3, 4]
print(max(numbers))  # 4
print(min(numbers))  # 1
```

**Output:**

```
4
1
```

1. `str()`: Converts an object into a string.

```
num = 123
print(str(num))  # '123'
```

**Output:**

```
'123'
```

1. `input()`: Reads user input from the console.

```
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

**Example Output:**

```
Enter your name: Alice
Hello, Alice!
```

1. `abs()`: Returns the absolute value of a number.

```
print(abs(-10))  # 10
```

**Output:**

```
10
```

---

## 3. Lambda Functions (Anonymous Functions)

Python also supports **lambda functions**, which are small anonymous functions defined with the `lambda` keyword. These are often used for short, simple operations.

### Syntax:

```
lambda arguments: expression
```

### Example:

```
multiply = lambda x, y: x * y
print(multiply(2, 3))  # 6
```

**Output:**

```
6
```

---

## 4. Recursion in Functions

A function that calls itself is known as a **recursive function**. Recursion is used when a problem can be divided into smaller subproblems of the same type.

### Example of Recursion (Factorial Calculation):

```
def factorial(n):
    if n == 0:  # Base case
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))  # 120
```

**Output:**

```
120
```

## Summary of Functions:

1. **User-Defined Functions**:
   - Defined using `def function_name()`.
   - Can take parameters and return values.
   - Supports default arguments and multiple return values.
2. **Built-in Functions**:
   - Predefined in Python.
   - Cover common tasks like mathematical operations, string manipulations, etc.
   - Examples: `print()`, `len()`, `sum()`, `type()`, etc.
3. **Lambda Functions**:
   - Small, anonymous functions defined with `lambda`.
   - Ideal for short, one-time operations.
4. **Recursion**:
   - A function that calls itself to solve problems by breaking them into smaller subproblems.

Functions in Python provide a clean, efficient, and reusable way to write code, making it easy to manage and debug.



## Math Module in Python

The **math** module in Python provides a set of functions and constants that help with mathematical operations. These operations range from basic arithmetic to advanced mathematical calculations like trigonometry, logarithms, and constants like π (pi) and e (Euler's number).

To use the math module, you need to import it first using:

```
import math
```

Once imported, you can access its functions and constants.

---

## Common Functions and Constants in the Math Module

### 1. Basic Mathematical Operations

- `math.sqrt(x)`: Returns the square root of `x`.

```
import math
result = math.sqrt(16)
print(result)  # 4.0
```

**Output:**

```
4.0
```

- `math.pow(x, y)`: Returns `x` raised to the power of `y`.

```
result = math.pow(2, 3)
print(result)  # 8.0
```

**Output:**

```
8.0
```

### 2. Mathematical Constants

- `math.pi`: The mathematical constant π (approx. 3.14159).

```
print(math.pi)  # 3.141592653589793
```

**Output:**

```
3.141592653589793
```

- `math.e`: The mathematical constant e (approx. 2.71828).

```
print(math.e)  # 2.718281828459045
```

**Output:**

```
2.718281828459045
```

## 3. Trigonometric Functions

- `math.sin(x)`: Returns the sine of `x` (x is in radians).

```
result = math.sin(math.pi / 2)  # sin(90 degrees)
print(result)  # 1.0
```

**Output:**

```
1.0
```

- `math.cos(x)`: Returns the cosine of `x` (x is in radians).

```
result = math.cos(math.pi)  # cos(180 degrees)
print(result)  # -1.0
```

**Output:**

```
-1.0
```

- `math.tan(x)`: Returns the tangent of `x` (x is in radians).

```
result = math.tan(math.pi / 4)  # tan(45 degrees)
print(result)  # 1.0
```

**Output:**

```
1.0
```

## 4. Logarithmic Functions

- `math.log(x)`: Returns the natural logarithm (base e) of `x`.

```
result = math.log(10)
print(result)  # 2.302585092994046
```

**Output:**

```
2.302585092994046
```

- `math.log10(x)`: Returns the base-10 logarithm of `x`.

```
result = math.log10(100)
print(result)  # 2.0
```

**Output:**

```
2.0
```

- `math.log2(x)`: Returns the base-2 logarithm of `x`.

```
result = math.log2(8)
print(result)  # 3.0
```

**Output:**

```
3.0
```

## 5. Factorial

- `math.factorial(x)`: Returns the factorial of `x` (x must be a non-negative integer).

```
result = math.factorial(5)
print(result)  # 120
```

**Output:**

```
120
```

## 6. Rounding Functions

- `math.floor(x)`: Returns the largest integer less than or equal to `x` (rounds down).

```
result = math.floor(4.7)
print(result)  # 4
```

**Output:**

```

```

- `math.ceil(x)`: Returns the smallest integer greater than or equal to `x` (rounds up).

```
result = math.ceil(4.3)
print(result)  # 5
```

**Output:**

```
5
```

- `math.trunc(x)`: Returns the integer part of `x`, truncating the decimal part.

```
result = math.trunc(4.8)
print(result)  # 4
```

**Output:**

```
4
```

## 7. Hyperbolic Functions

- `math.sinh(x)`: Returns the hyperbolic sine of `x`.

```
result = math.sinh(1)
print(result)  # 1.1752011936438014
```

**Output:**

```
1.1752011936438014
```

- `math.cosh(x)`: Returns the hyperbolic cosine of `x`.

```
result = math.cosh(1)
print(result)  # 1.5430806348152437
```

**Output:**

```
1.5430806348152437
```

## 8. Angle Conversion Functions

- `math.degrees(x)`: Converts angle `x` from radians to degrees.

```
result = math.degrees(math.pi)
print(result)  # 180.0
```

**Output:**

```
180.0
```

- `math.radians(x)`: Converts angle `x` from degrees to radians.

```
result = math.radians(180)
print(result)  # 3.141592653589793
```

**Output:**

```
3.141592653589793
```

## 9. Greatest Common Divisor (GCD)

- `math.gcd(x, y)`: Returns the greatest common divisor of `x` and `y`.

```
result = math.gcd(36, 60)
print(result)  # 12
```

**Output:**

```
12
```

---

## Summary of Common Math Module Functions:

1. **Mathematical Constants**:
   - `math.pi`: Value of π.
   - `math.e`: Euler's number.
2. **Mathematical Operations**:
   - `math.sqrt(x)`: Square root.
   - `math.pow(x, y)`: Power operation.
3. **Trigonometry**:
   - `math.sin(x)`, `math.cos(x)`, `math.tan(x)`: Sine, Cosine, and Tangent functions.
4. **Logarithms**:
   - `math.log(x)`: Natural logarithm.

- `math.log10(x)`: Base-10 logarithm.
- `math.log2(x)`: Base-2 logarithm.

5. **Factorial**:
   - `math.factorial(x)`: Factorial of x.
6. **Rounding Functions**:
   - `math.floor(x)`, `math.ceil(x)`, `math.trunc(x)`.
7. **Hyperbolic Functions**:
   - `math.sinh(x)`, `math.cosh(x)`.
8. **Angle Conversion**:
   - `math.degrees(x)`, `math.radians(x)`.
9. **GCD**:
   - `math.gcd(x, y)`.

The `math` module provides efficient tools for performing mathematical operations and is very useful when working with numbers in Python.



## List Comprehensions in Python

List comprehensions provide a concise way to create lists in Python. They consist of an expression followed by a `for` loop, and optionally, an `if` condition. List comprehensions allow for the creation of new lists by applying an expression to each element of an existing iterable (like a list, range, or string) in a single line of code.

## Basic Syntax

```
[expression for item in iterable]
```

- **expression**: An operation or value that is computed and added to the new list for each element of the iterable.

- **item**: The variable representing each element in the iterable.
- **iterable**: Any Python iterable (e.g., a list, range, etc.) that you want to iterate over.

## Examples of List Comprehensions

### 1. Simple List Comprehension

Create a list of squares of numbers from 0 to 4:

```
squares = [x**2 for x in range(5)]
print(squares)
```

**Output:**

```
[0, 1, 4, 9, 16]
```

### 2. Using `if` Condition in List Comprehension

List comprehensions can also include an `if` condition to filter elements.

For example, to create a list of even numbers from 0 to 9:

```
even_numbers = [x for x in range(10) if x % 2 == 0]
print(even_numbers)
```

**Output:**

```
[0, 2, 4, 6, 8]
```

### 3. Using `if` and `else` in List Comprehension

You can use both `if` and `else` in a list comprehension to apply a condition and return different values.

For example, create a list where numbers are replaced by "even" or "odd":

```
result = ['even' if x % 2 == 0 else 'odd' for x in range(5)]
print(result)
```

**Output:**

```
['even', 'odd', 'even', 'odd', 'even']
```

## 4. Nested List Comprehension

List comprehensions can be nested inside each other. For example, to create a list of pairs from two lists:

```
pairs = [(x, y) for x in range(3) for y in range(2)]
print(pairs)
```

**Output:**

```
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

## 5. List Comprehension with Functions

You can apply a function to each element while creating a new list. For example, using a built-in `str()` function to convert numbers to strings:

```
numbers = [1, 2, 3, 4]
string_numbers = [str(num) for num in numbers]
print(string_numbers)
```

**Output:**

```
['1', '2', '3', '4']
```

## 6. Flatten a List of Lists Using List Comprehension

If you have a nested list and want to flatten it into a single list, you can use a list comprehension:

```
nested_list = [[1, 2], [3, 4], [5, 6]]
flat_list = [item for sublist in nested_list for item in subl
ist]
print(flat_list)
```

**Output:**

```
[1, 2, 3, 4, 5, 6]
```

## Advantages of List Comprehension

1. **Conciseness**: List comprehensions provide a shorter and more readable way to create lists.
2. **Performance**: They are generally more efficient than traditional loops for creating lists.
3. **Functional Style**: They enable a more declarative approach to creating lists, emphasizing what needs to be done rather than how.

## When to Use List Comprehensions

- When you need to create a new list by applying an operation or transformation to each element of an existing iterable.
- When you want to filter elements based on a condition.
- When you want a more concise, readable way to create lists compared to using loops.

## Avoid Overuse

While list comprehensions are concise, they can become difficult to understand if the expression or condition becomes too complex. In such cases, traditional loops might be more readable.

## Summary of List Comprehensions

- List comprehensions allow you to create lists in a concise and readable way.
- You can use expressions, conditions (`if`, `else`), and even nested loops inside a list comprehension.
- They are ideal for applying operations to each element of an iterable or filtering elements based on a condition.

## Lambda, Map, and Filter in Python

### 1. Lambda Functions

A `lambda` function is a small anonymous function that can have any number of arguments but only one expression. The expression is evaluated and returned when the function is called.

- **Syntax**:

```
lambda arguments: expression
```

- **Example**:

  A `lambda` function to add two numbers:

```
add = lambda x, y: x + y
print(add(2, 3))  # Output: 5
```

  In this example, the `lambda` function takes two arguments `x` and `y`, and returns their sum.

### Use cases of lambda functions:

- Used when you need a simple function for a short period of time and do not want to formally define it using `def`.
- Can be used in places where functions are expected as arguments, such as with `map()`, `filter()`, and `sorted()`.

---

### 2. Map Function

The `map()` function applies a given function to all items in an iterable (such as a list, tuple, etc.) and returns an iterator that produces the results.

- **Syntax**:

```
map(function, iterable)
```

- **Parameters**:
  - `function`: The function to apply to each item in the iterable.
  - `iterable`: The iterable (list, tuple, etc.) whose items are to be processed.
- **Example**:

  To square each number in a list using `map()` and a `lambda` function:

```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16, 25]
```

  Here, `map()` applies the `lambda` function (which squares each element) to each item in the `numbers` list.
- **Use cases**:
  - Apply the same operation to each element of a list or iterable.
  - Can be used with functions and `lambda` expressions to transform data.

---

## 3. Filter Function

The `filter()` function is used to filter elements from an iterable based on a condition defined by a function. It returns an iterator that contains only the elements for which the function returns `True`.

- **Syntax**:

```
filter(function, iterable)
```

- **Parameters**:

- `function`: The function that defines the filtering condition. It should return a boolean value (`True` or `False`).
- `iterable`: The iterable (list, tuple, etc.) to be filtered.

- **Example**:

  To filter out all odd numbers from a list using `filter()` and a `lambda` function:

  ```
  numbers = [1, 2, 3, 4, 5, 6]
  even_numbers = filter(lambda x: x % 2 == 0, numbers)
  print(list(even_numbers))  # Output: [2, 4, 6]
  ```

  In this example, the `lambda` function checks if a number is even (`x % 2 == 0`), and `filter()` includes only the elements that satisfy this condition.

- **Use cases**:
  - Filter data based on some condition.
  - Can be used to remove unwanted elements from an iterable (e.g., odd numbers, empty strings, etc.).

---

## Comparison: Lambda, Map, and Filter

- **Lambda**: It's a simple anonymous function that can be used for one-line expressions.
- **Map**: Applies a function to all items in an iterable and returns a new iterable with the transformed items.
- **Filter**: Filters elements from an iterable based on a function's condition, returning only those that meet the condition.

---

## More Examples

### 1. Using Lambda with Map:

Squaring a list of numbers using `map()`:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x**2, numbers)
print(list(squared_numbers))  # Output: [1, 4, 9, 16, 25]
```

## 2. Using Lambda with Filter:

Filtering out negative numbers:

```
numbers = [-1, -2, 3, 4, -5]
positive_numbers = filter(lambda x: x > 0, numbers)
print(list(positive_numbers))  # Output: [3, 4]
```

## 3. Using Lambda with Both Map and Filter:

Transform a list by squaring only the positive numbers:

```
numbers = [-1, 2, -3, 4, 5]
positive_squared_numbers = map(lambda x: x**2, filter(lambda
x: x > 0, numbers))
print(list(positive_squared_numbers))  # Output: [4, 16, 25]
```

Here, `filter()` first filters out the negative numbers, and then `map()` squares the remaining positive numbers.

---

## Summary

- **Lambda** functions are anonymous and are often used for short operations.
- **Map** applies a function to every item in an iterable and returns a new iterable with the results.
- **Filter** filters an iterable based on a condition, returning only elements that satisfy that condition.

## Exception Handling in Python

Exception handling in Python allows you to deal with errors gracefully and prevent program crashes. Python uses a mechanism called **try-except** to handle exceptions. By catching exceptions, your program can continue running even when an error occurs, and you can provide helpful feedback to the user.

## Basic Syntax of Exception Handling:

```
try:
    # Code that may raise an exception
    result = 10 / 0  # This will raise a ZeroDivisionError
except ExceptionType:
    # Code to handle the exception
    print("An error occurred!")
```

- `try` **block**: This is where you write the code that might raise an exception.
- `except` **block**: This is where you catch and handle the exception.
- `ExceptionType`: This is the type of exception you want to catch (e.g., `ZeroDivisionError`, `FileNotFoundError`). You can also catch all exceptions using `except` alone, but it's better to catch specific exceptions to avoid masking bugs.

---

## Types of Exceptions in Python:

Some common Python exceptions are:

- `ZeroDivisionError`: Raised when you try to divide by zero.
- `FileNotFoundError`: Raised when a file is not found.
- `ValueError`: Raised when a function receives an argument of the correct type, but an inappropriate value.
- `IndexError`: Raised when you try to access an invalid index in a list.
- `KeyError`: Raised when you try to access a dictionary with a key that does not exist.

---

## Example 1: Basic Try-Except Block

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
    print("Please enter a valid integer.")
```

- If the user enters zero, a `ZeroDivisionError` is raised.
- If the user enters something that cannot be converted to an integer, a `ValueError` is raised.

---

## Example 2: Catching Multiple Exceptions

You can catch multiple exceptions in a single `try-except` block by specifying different `except` blocks for each type of error.

```
try:
    # Trying to open a file that may not exist
    file = open('nonexistent_file.txt', 'r')
except FileNotFoundError:
    print("The file does not exist!")
except IOError:
    print("There was an I/O error!")
```

In this case:

- If the file doesn't exist, a `FileNotFoundError` is raised.
- If there's any other I/O issue, an `IOError` will be caught.

---

## Example 3: Catching All Exceptions

You can catch all exceptions with a general `except` block. However, this is not recommended because it may hide unexpected errors, making debugging

difficult.

```
try:
    # Some code that may raise an error
    result = 10 / 0
except:
    print("An error occurred!")  # Will catch any exception
```

While this catches all exceptions, it is best to catch specific exceptions to handle errors more meaningfully.

---

## Example 4: Using `else` with Exception Handling

An `else` block can be used to specify code that should run if no exceptions are raised in the `try` block.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
    print("Please enter a valid integer.")
else:
    print(f"The result is: {result}")
```

- If no exception occurs, the code in the `else` block will run.

---

## Example 5: Using `finally` Block

The `finally` block allows you to define code that will run regardless of whether an exception was raised or not. It is typically used for cleanup actions, such as closing files or releasing resources.

```
try:
    file = open('example.txt', 'r')
```

```
    # Some code that could raise an exception
except FileNotFoundError:
    print("File not found!")
finally:
    print("This will always run.")
    # Closing the file if it was opened
    file.close()
```

- The `finally` block runs after the `try` block, whether or not an exception occurred. It ensures that necessary cleanup, like closing files or releasing resources, is done.

---

## Raising Exceptions with `raise`

You can raise exceptions manually using the `raise` keyword. This is useful if you want to enforce certain conditions in your code.

```
def check_positive(num):
    if num < 0:
        raise ValueError("Number must be positive!")
    return num

try:
    check_positive(-1)
except ValueError as e:
    print(f"Error: {e}")
```

In this example:

- A `ValueError` is raised if the number is negative.

---

## Custom Exceptions

You can define your own exceptions by subclassing the built-in `Exception` class.

```
class CustomError(Exception):
    pass
```

```
try:
    raise CustomError("This is a custom error!")
except CustomError as e:
    print(f"Caught an error: {e}")
```

In this case, `CustomError` is a user-defined exception, and we can catch it just like any other exception.

---

## Summary of Exception Handling Keywords:

- `try`: Contains code that may raise an exception.
- `except`: Catches and handles exceptions.
- `else`: Runs if no exception was raised.
- `finally`: Always runs, used for cleanup.
- `raise`: Raises an exception explicitly.

---

## Why Use Exception Handling?

- **Error Management**: It helps manage errors in a controlled way.
- **Graceful Degradation**: The program can recover from errors and continue running.
- **User Feedback**: Provides meaningful messages to the user when things go wrong.



## Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects and classes. Python is an object-oriented language, meaning it supports concepts like **classes**, **objects**, **inheritance**, **abstraction**, and **polymorphism**.

## Key Concepts of OOP:

1. **Class**: A blueprint for creating objects (instances). It defines the properties (attributes) and behaviors (methods) that the objects will have.
2. **Object**: An instance of a class. It is a collection of data (attributes) and functions (methods) that operate on the data.
3. **Inheritance**: A way to create a new class by using details from an existing class, inheriting its methods and attributes.
4. **Abstraction**: Hiding the complex implementation details of a system and exposing only the necessary features.
5. **Polymorphism**: The ability to use a method in multiple forms, typically by overriding or overloading methods in subclasses.

---

## 1. Classes in Python

A **class** in Python is a template for creating objects, and it defines a set of attributes and behaviors.

### Example: Defining a Class

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start_engine(self):
        print(f"{self.make} {self.model} engine started.")

    def stop_engine(self):
        print(f"{self.make} {self.model} engine stopped.")
```

- `__init__` **method**: A special method used to initialize the object. It is automatically called when an object is created.
- `self`: Refers to the current instance of the class. It is used to access the attributes and methods of the class.

### Creating an Object

```
my_car = Car("Toyota", "Corolla", 2020)
my_car.start_engine()  # Output: Toyota Corolla engine starte
d.
```

---

## 2. Inheritance in Python

**Inheritance** allows a class to inherit the attributes and methods of another class. This promotes code reuse and can be used to create specialized classes.

### Example: Inheritance

```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start_engine(self):
        print(f"{self.make} {self.model} engine started.")

class Car(Vehicle):
    def __init__(self, make, model, year):
        super().__init__(make, model)  # Calls the __init__ o
f the parent class
        self.year = year

    def display_info(self):
        print(f"{self.year} {self.make} {self.model}")

# Creating an object of Car class, which inherits from Vehicl
e
my_car = Car("Toyota", "Corolla", 2020)
my_car.start_engine()  # Inherited method
my_car.display_info()  # Car's own method
```

- `super().__init__()`: This calls the `__init__` method of the parent class (in this case, `Vehicle`), allowing the `Car` class to inherit its properties.

### Output:

```
Toyota Corolla engine started.
2020 Toyota Corolla
```

---

## 3. Abstraction in Python

**Abstraction** refers to hiding the implementation details and showing only the essential features of an object. This is achieved using **abstract classes** and **abstract methods**.

### Abstract Class and Method

An abstract class is a class that cannot be instantiated and typically contains one or more abstract methods. Abstract methods are methods that are declared but contain no implementation in the abstract class; they must be implemented in a subclass.

To use abstraction in Python, we need to import the **abc (Abstract Base Class)** module.

### Example: Abstraction

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Animal is now an abstract class
    @abstractmethod
    def sound(self):
        pass  # Abstract method, to be implemented by subclas
ses

class Dog(Animal):
    def sound(self):
        print("Bark")
```

```
class Cat(Animal):
    def sound(self):
        print("Meow")

# animal = Animal()  # This will raise an error as Animal is
abstract
dog = Dog()
dog.sound()  # Output: Bark
cat = Cat()
cat.sound()  # Output: Meow
```

- `ABC`: This is a class from the `abc` module that makes the class an abstract class.
- `@abstractmethod`: This decorator is used to define methods that must be implemented in subclasses.

---

## 4. Polymorphism in Python

**Polymorphism** allows different classes to be treated as instances of the same class through inheritance. The most common use of polymorphism is method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.

## Method Overriding (Polymorphism)

```
class Animal:
    def speak(self):
        print("Animal is speaking")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
```

```
        print("Cat meows")

# Polymorphism in action
animals = [Dog(), Cat()]

for animal in animals:
    animal.speak()  # Each object calls its own speak method
```

## Output:

```
Dog barks
Cat meows
```

- In this example, the `speak()` method is overridden in both the `Dog` and `Cat` classes. Despite the different implementations, you can treat both `Dog` and `Cat` objects as instances of the `Animal` class, and they respond with their own specific implementations of `speak()`.

---

## Summary of OOP Concepts:

1. **Class**: A blueprint for objects, defining attributes and methods.
2. **Object**: An instance of a class containing the actual data.
3. **Inheritance**: A mechanism to create a new class from an existing class, inheriting its properties and behaviors.
4. **Abstraction**: Hiding complex details and exposing only essential features. This is implemented using abstract classes and methods.
5. **Polymorphism**: The ability for different classes to provide their own implementation of a method (method overriding), allowing them to be used interchangeably.

OOP principles help in writing reusable, maintainable, and flexible code, and they are foundational concepts in Python and many other programming languages.